



ML 08: K-MEANS, PCA DETAILS

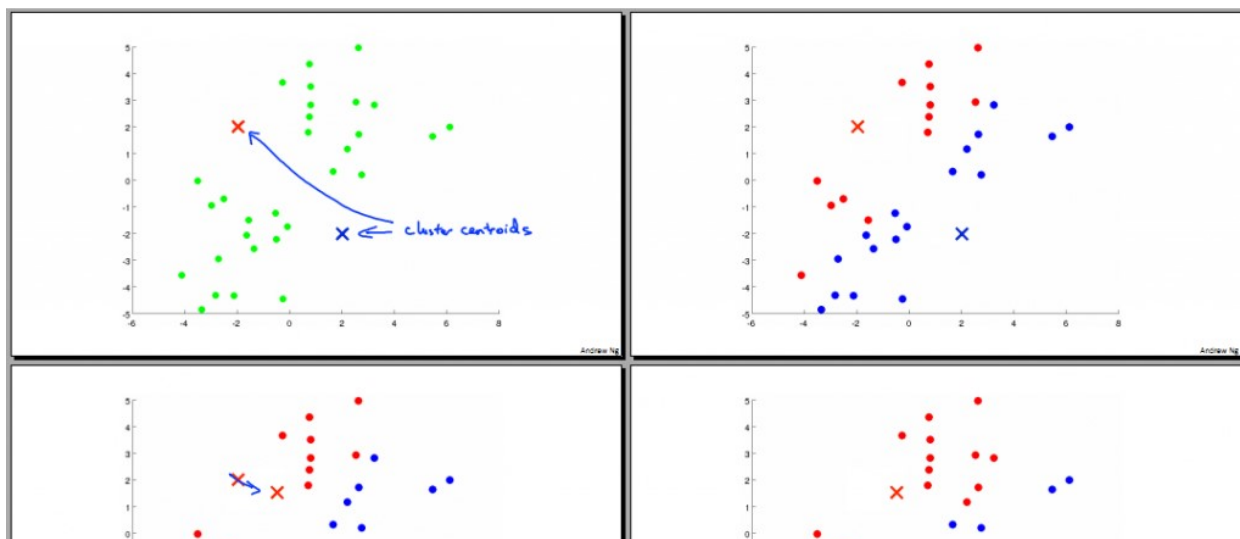
이번시간에는 *PCA* 와 *clustering* 을 배운다. *PCA* 가 어떻게 돌아가는지 알기위해 *covariance matrix*, *eigen decomposition*, *singular value decomposition* 등의 배경지식도 익혀보자. ~~K-means 는 거들뿐

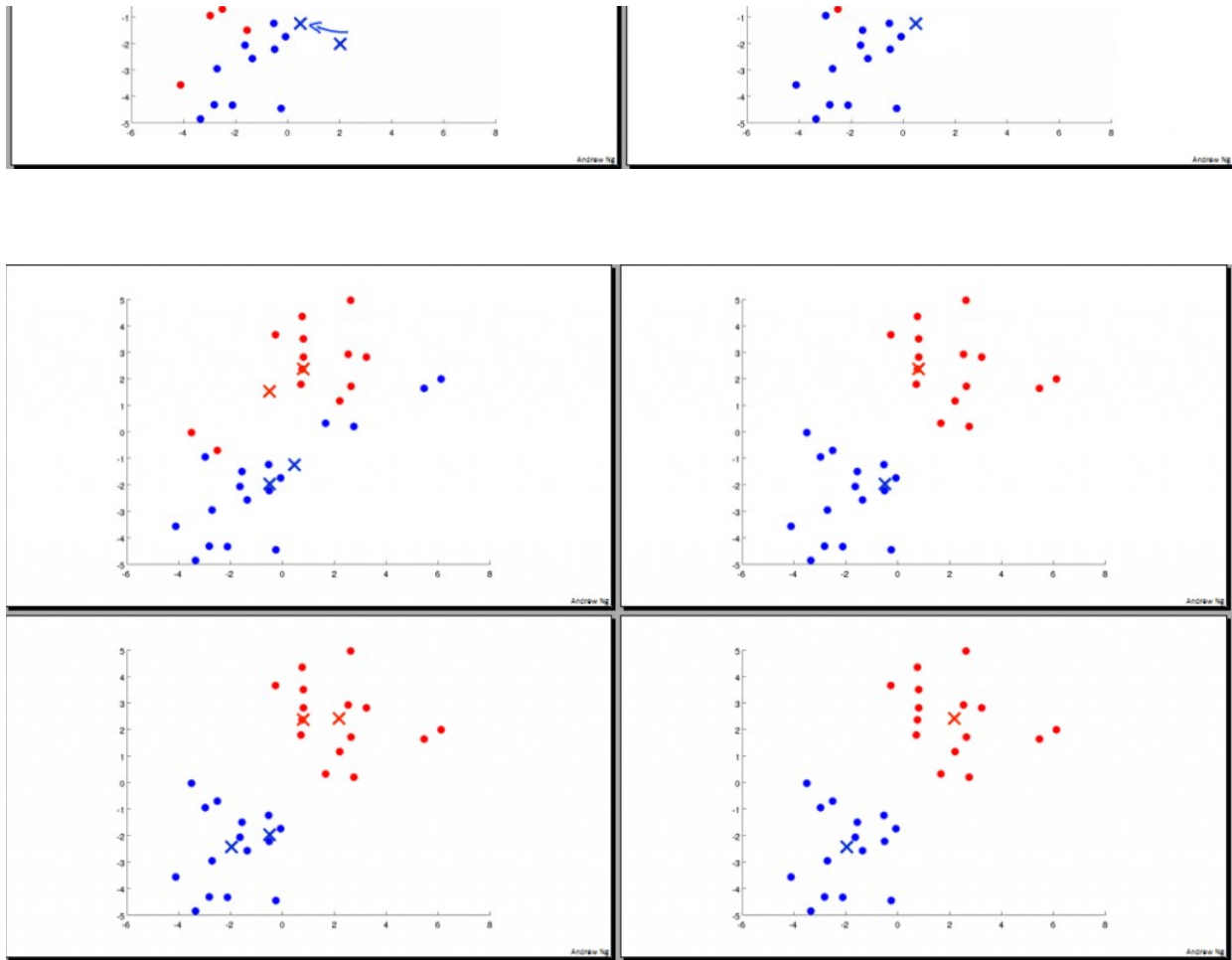
Unsupervised Learning Intro

clustering 은 다양한 분야에 활용할 수 있다.

- Market Segmentation
- Social Network Analysis
- Organize Computing Clusters
- Astronomical Data Analysis

K-Means





(<http://blog.csdn.net/linuxcunt>)

랜덤한 위치에 *centroid* 를 잡고, 가까운 점들을 색칠 한뒤 그 점들의 중심으로 *centroid* 를 옮겨가면서 집단을 만들어 낸다.

k-means 의 인풋은 *centroid* 의 수인 k 와 x_1, x_2, \dots, x_m 의 트레이닝 셋이다. 구체적인 알고리즘은

K-means algorithm

μ_1 μ_2
x x

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K \in \mathbb{R}^n$

Repeat {

Cluster
assignment
step

for $i = 1$ to m

$c^{(i)} :=$ index (from 1 to K) of cluster centroid
closest to $x^{(i)}$

$\min_k \|x^{(i)} - \mu_k\|^2$
 \downarrow
 $c^{(i)}$

Move

for $k = 1$ to K

$\mu_k :=$ average (mean) of points assigned to cluster k

centroids μ_k :- average (mean) of points assigned to cluster k

$x^{(1)}, x^{(5)}, x^{(6)}, x^{(10)} \rightarrow c^{(1)}=2, c^{(5)}=2, c^{(6)}=2, c^{(10)}=2$

$\mu_2 = \frac{1}{4} [x^{(1)} + x^{(5)} + x^{(6)} + x^{(10)}] \in \mathbb{R}^n$

(<http://blog.csdn.net/linuxcunt>)

매 이터레이션마다 i to m 까지 루프를 돌면서 클러스터링된 원소들의 배열인 $c^{(i)}$ 에 1 to K 사이의 값을 넣는다. 이때 $c^{(i)}$ 에 삽입될 값은, 해당 원소로부터 가장 가까운 *centroid*의 인덱스다.

$c^{(i)}$ is index of cluster $(1, \dots, K)$ to which example $x^{(i)}$ is currently assigned

따라서 각 원소로부터의 거리를 최소로 하는 k 에 대해 $c^{(i)} = k$ 다.

$$\min_k \|x^{(i)} - \mu_k\|$$

$$\Rightarrow c^{(i)} = k$$

Optimization Objective

K-means optimization objective

$\rightarrow c^{(i)}$ = index of cluster $(1, 2, \dots, K)$ to which example $x^{(i)}$ is currently assigned

$\rightarrow \mu_k$ = cluster centroid k ($\mu_k \in \mathbb{R}^n$)

$\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

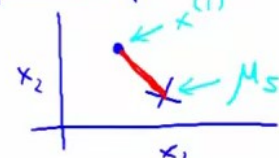
$x^{(i)} \rightarrow 5 \quad c^{(i)} = 5 \quad \mu_{c^{(i)}} = \mu_5$

Optimization objective:

$$\rightarrow J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

$\rightarrow \min_{c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

Distortion



(<http://blog.csdn.net/linuxcunt>)

k -means 에서 최소화 하려는 *cost function* 은

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|$$

다시 말해서 각 점에서 *centroid* 까지의 거리를 최소화 하는 것이 목표다.

$$\min_{c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$$

이 함수 J 를 다른말로는 *distortion function* 이라 부른다. 알고리즘을 다시 보면

K-means algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K \in \mathbb{R}^n$

Repeat {

Cluster assignment step
Minimize $J(\dots)$ w.r.t. $c^{(1)}, c^{(2)}, \dots, c^{(m)}$ ←
(holding μ_1, \dots, μ_K fixed)

for $i = 1$ to m
 $c^{(i)} :=$ index (from 1 to K) of cluster centroid
 closest to $x^{(i)}$

move centroid
for $k = 1$ to K
 $\mu_k :=$ average (mean) of points assigned to cluster k

} minimize $J(\dots)$ w.r.t. μ_1, \dots, μ_K

(<http://blog.csdn.net/linuxcunt>)

(1) *clustering assignment step*에서는 μ 를 고정시키고 $c^{(i)}$ 에 대해서 J 를 최소화 한다.

(2) *move centroid step*에서는 $c^{(i)}$ 를 고정시키고 μ 에 대해서 J 를 최소화 한다.

Random Initialization

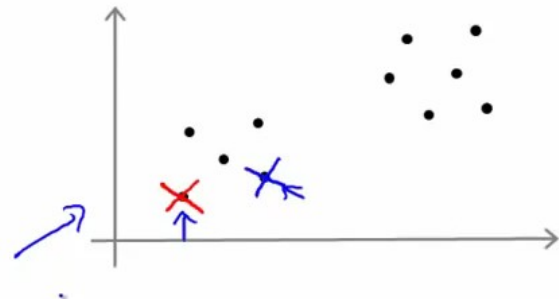
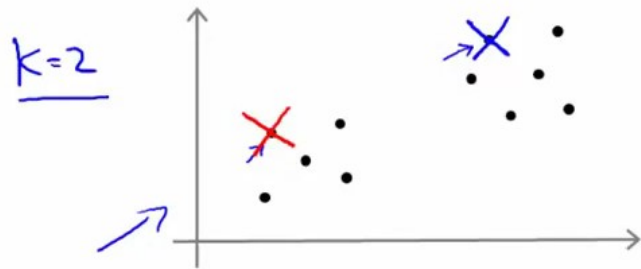
Random initialization

Should have $K < m$

Randomly pick K training examples.

Set μ_1, \dots, μ_K equal to these K examples.

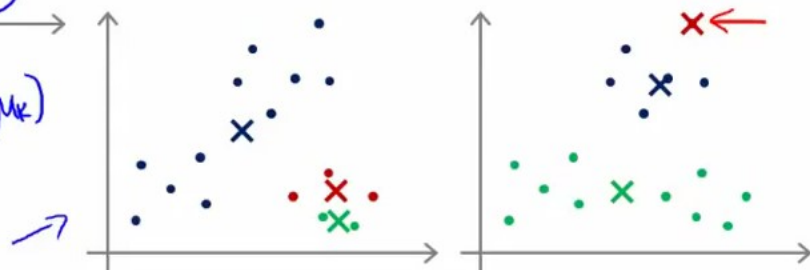
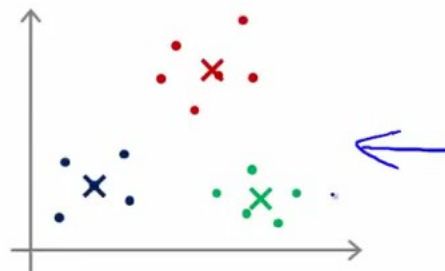
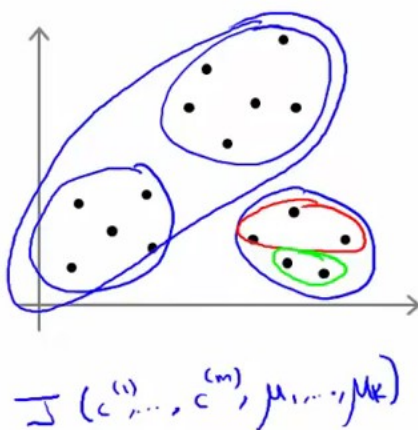
$$\begin{aligned}\mu_1 &= x^{(i)} \\ \mu_2 &= x^{(j)} \\ &\vdots\end{aligned}$$



(<http://blog.csdn.net/linuxcumt>)

위쪽 예제는 *centroid*의 랜덤 초기화에서 좋게 배치된 경우이고, 아래쪽은 운이 나쁜 경우를 설명하는 그림이다. 이것이 설명하는 바는 *centroid*의 초기화에 따라 결과가 달라질 수 있다는 것이다. 아래 그림을 보면

Local optima



(<http://blog.csdn.net/linuxcunt>)

따라서 *local optima* 를 피하기 위해, 그리고 좋은 *clustering* 을 얻기 위해 *random initialization* 이용하여 *k-mean* 를 여러번 돌릴 수 있다.

Random initialization

```

For i = 1 to 100 {
    → Randomly initialize K-means.
    Run K-means. Get  $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K$ .
    Compute cost function (distortion)
    →  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$ 
}
  
```

Handwritten notes: A bracket on the loop is labeled "50 - 1000".

Pick clustering that gave lowest cost $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

Handwritten notes: "k=2-10" is written below the loop, and an arrow points to the cost function.

(<http://blog.csdn.net/linuxcunt>)

여러번 *k-mean* 를 돌려 얻은 J 에 대해 최소값을 가지는 J 를 이용해 클러스터링을 얻을 수 있다.

그러나 K 가 매우 크다면, *random initialization* 들어가는 계산 비용에 비해 별로 좋은 결과를 돌려주지 못할 것이다.

random initialization 을 하는 방법으로

```

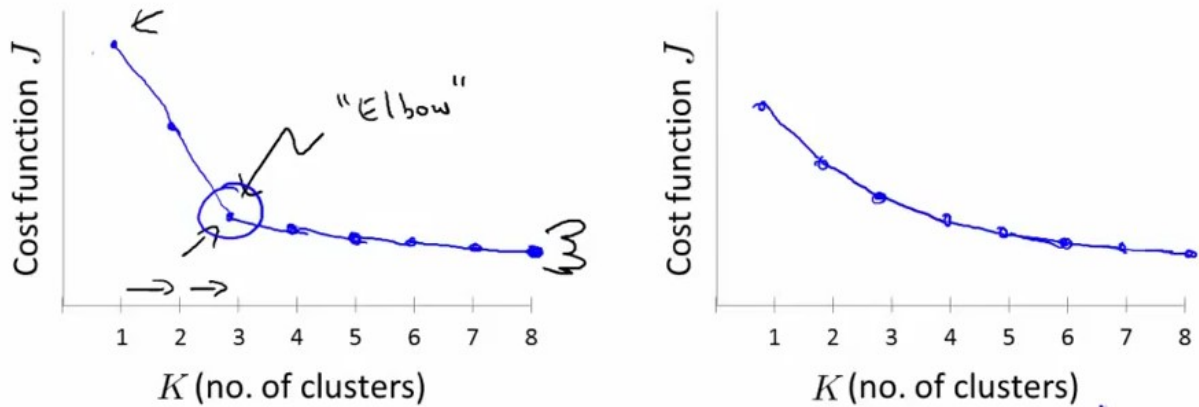
Pick  $k$  distinct random integers  $i_1, \dots, i_k$  from
 $\{1, \dots, m\}$  .
Set  $\mu_1 = x^{(i_1)}, \dots, \mu_k = x^{(i_k)}$ 
  
```

Choosing the Number of Cluster

K 값을 선택하기 위해 *Elbow method* 를 이용할 수 있다.

Choosing the value of K

Elbow method:

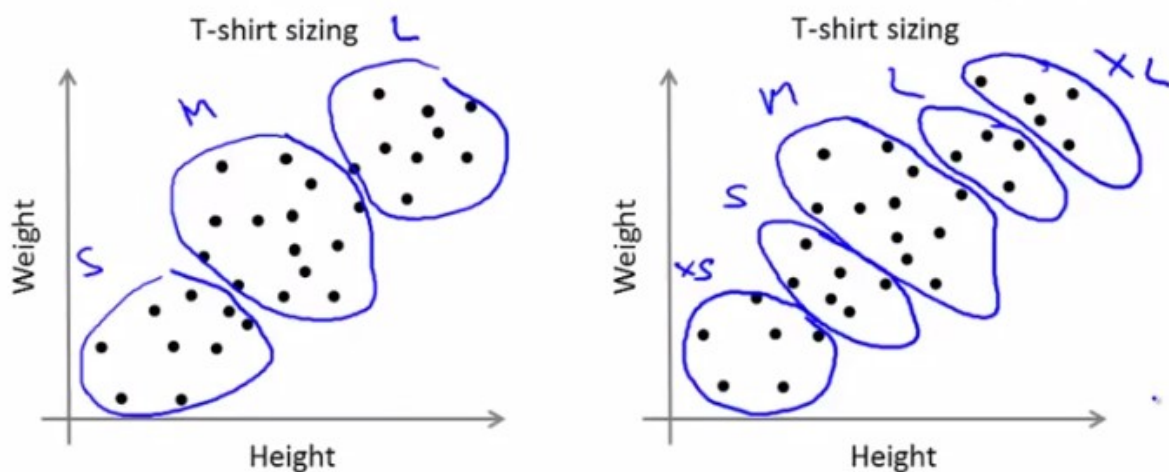


(<http://blog.csdn.net/linuxcunt>)

K 값의 변화에 따라 *distortion function* J 값이 급격히 감소하는 지점을 선택하는 방법이다. 그런데 J 값이 오른쪽 그림처럼 좀 애매하게 감소하면 어떻게 할까?

Sometimes, you are running K-means to get clusters to use for some later / downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose.

예를 들어서, 몸무게 / 키 에 따라 집단을 분류해 그에 맞추어 티셔츠를 만든다고 할 때 $K = 3 \text{ or } 5$ 에 대해서는 단순히 좋은 클러스터링을 얻는것은 물론 어떤 K 가 수지 타산이 더 맞을지 티셔츠 비즈니스적인 관점에서 생각을 해봐야 한다.



(<http://blog.csdn.net/abcjennifer>)

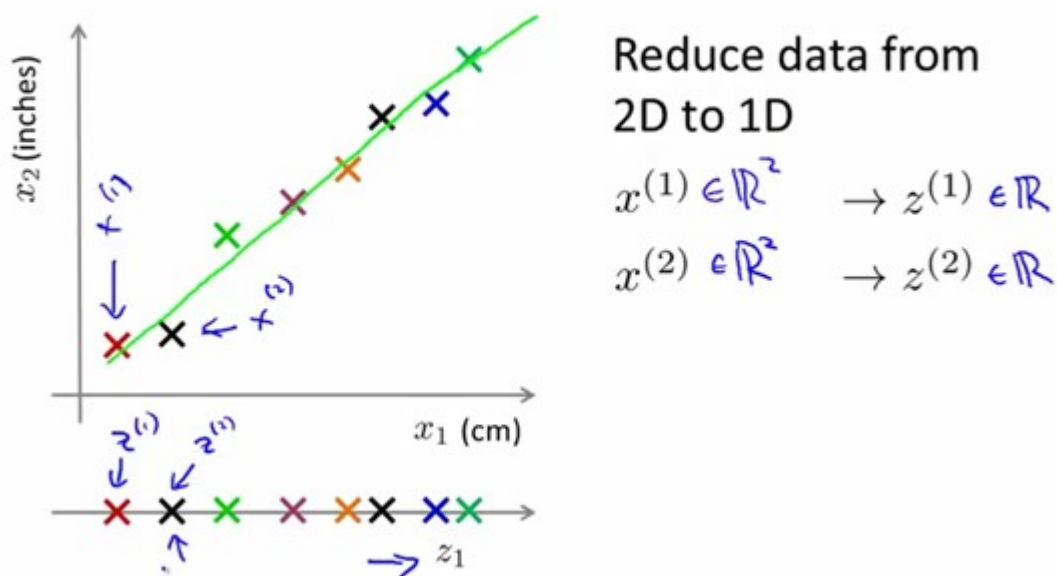
Dimensionality Reduction

이번엔 *unsupervised learning*의 또 다른 기법인 *dimensionality reduction*을 알아보자. 이 기법의 *motivation*은 2가지다.

- (1) Data Compression
- (2) Data Visualization

Data Compression

Data Compression



(<http://blog.csdn.net/linuxcuml>)

두 축을 보면 하나는 인치로, 다른 하나는 센치다. *highly redundant data*이기 때문에 하나의 차원으로 축소할 수 있다.

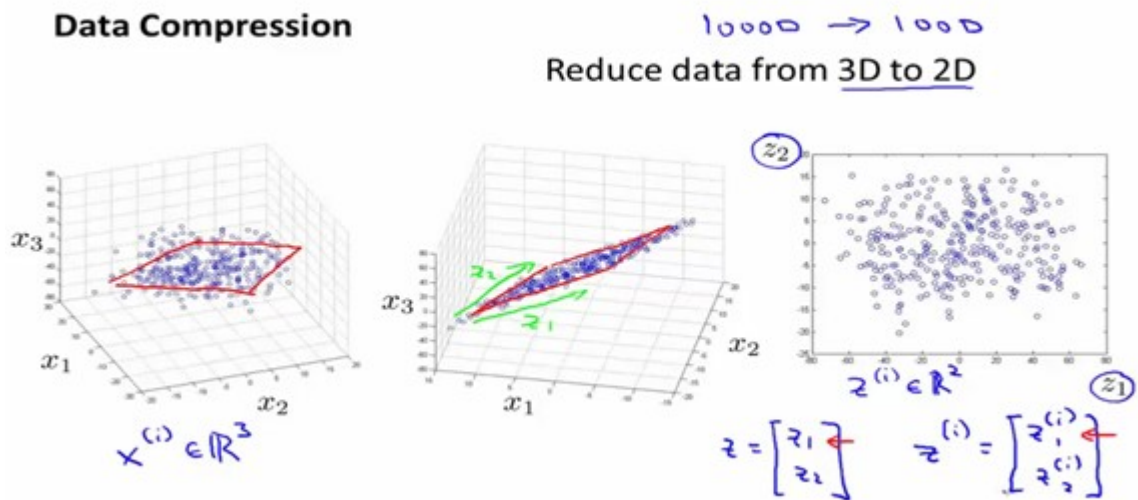
중복된 *feature*만 차원을 줄일 수 있는 것은 아니다

예를 들어 데이터의 한 축을 *pilot skill* 다른 축을 *pilot enjoyment*라 하고 두 *feature* 간 관계를 거의 직선으로 나타낼 수 있다고 하자. 이 새로운 직선을 *pilot aptitude*라 부르고 두개의 *feature*를 대신하는 새로운 *feature*로 사용할 수 있다.

feature 가 한 두개면 중복되는 것을 걸러내거나, 새로운 *feature* 로 만들기 쉬운데 만약 수백개라면 이것도 일이다.

위 그림을 다시 보면 x_1, x_2 를 z_1 으로 대신하고 있다. *feature* 의 수가 줄어든 것이다.

이렇게 차원을 줄이면 연산량이 줄어들고 전체 알고리즘의 성능도 빨라진다.

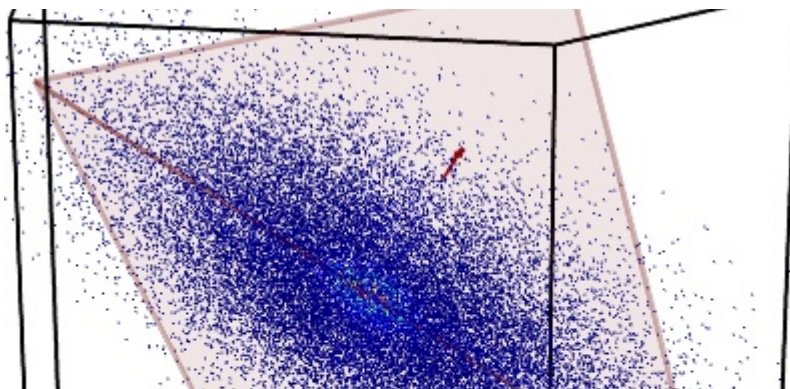


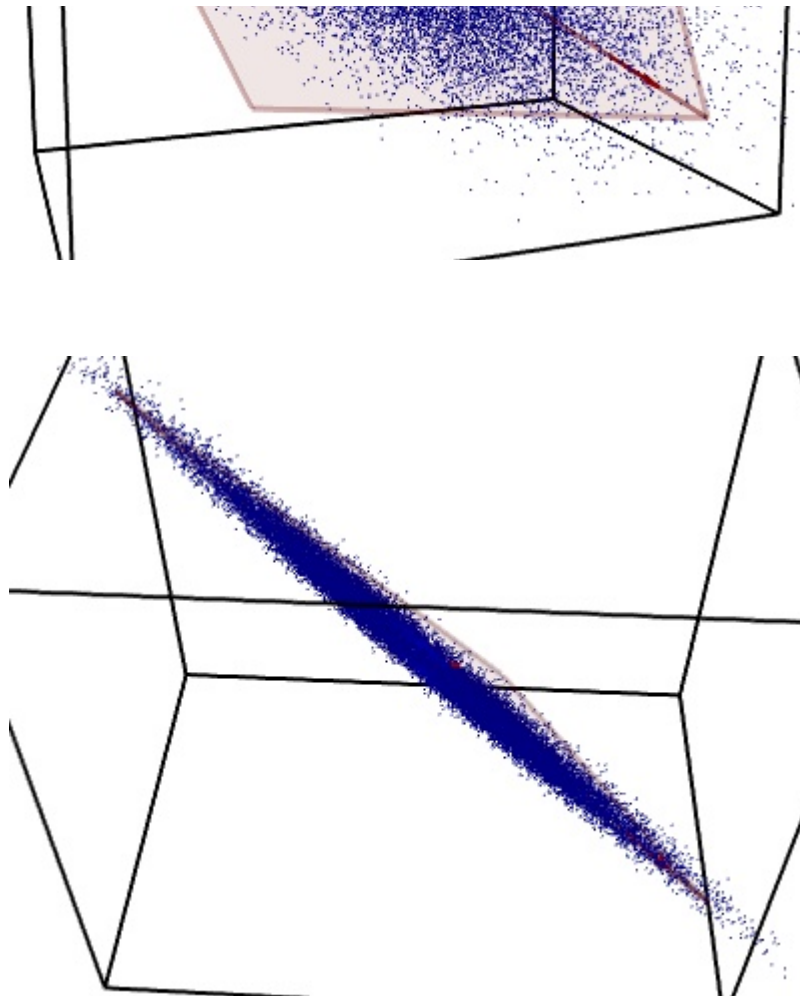
(<http://blog.csdn.net/linuxcunt>)

위 그림이 *dimensionality reduction* 에 대한 *intuition* 을 제공한다. 3차원의 데이터를 2차원의 평면으로 투영해 새로운 *feature set* 인 z 를 사용해 데이터를 나타낼 수 있다.

그림은 3차원 \rightarrow 2차원 이지만, 만약 10000 개를 1000 개를 줄일 수 있다면 어마어마한 중복을 줄일 수 있다.

조금 더 이해가 잘되는 3차원 그림을 가져오면





(<http://scipy-lectures.github.io>)

Data Visualization

dimensionality reduction 의 두 번째 *motivation* 은 바로 *data visualization* 이다.

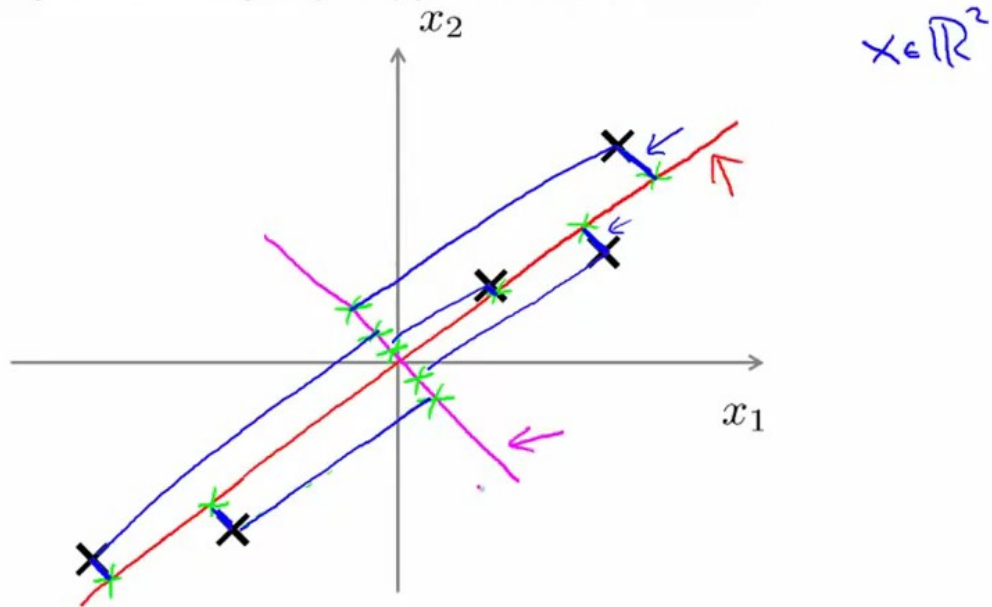
GDP, *Country size* 등 다양한 *feature* 500개를 2개로 줄여 그래프에 그려보면 데이터에 대한 어떤 *intuition* 을 얻을 수도 있다. 즉 데이터가 주로 어떤 종류의 *feature* 에 의해 많이 영향을 받는지 파악할 수 있다는 것이다.

강의에 나온 예제에서는 z_1 은 *country size / GDP*, z_2 는 *per person GDP* 였다.

visualization 을 위한 경우 2, 3 개 정도로 차원을 줄일 수 있다.

Principal Component Analysis Problem Formulation

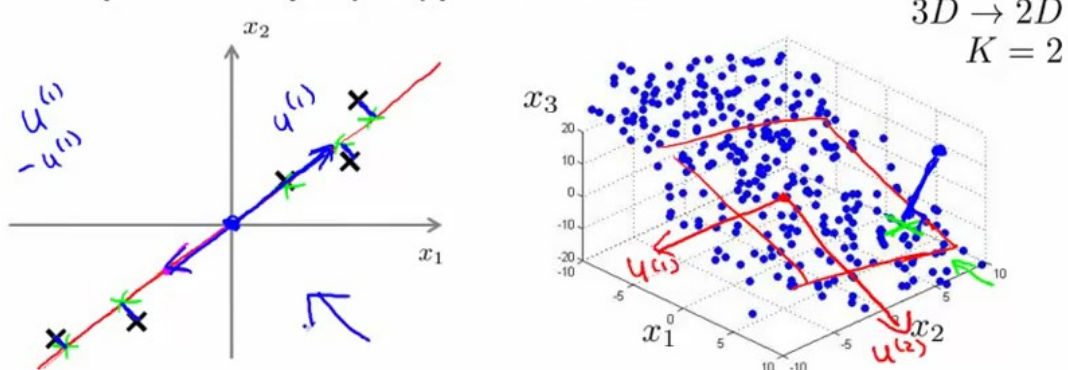
Principal Component Analysis (PCA) problem formulation



(<http://blog.csdn.net/linuxcumt>)

2개의 feature x_1, x_2 를 하나로 줄인다고 하자. 이 경우 *projection error* (파란 선의 길이) 를 최소로 하는 선 (빨강)을 찾으려고 할 것이다. 반면 자주색 선의 경우 *projection error* 가 가장 큰 선이라 볼 수있다.

Principal Component Analysis (PCA) problem formulation



Reduce from 2-dimension to 1-dimension: Find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error.

Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, \dots, u^{(k)}$

onto which to project the data, so as to minimize the projection error.

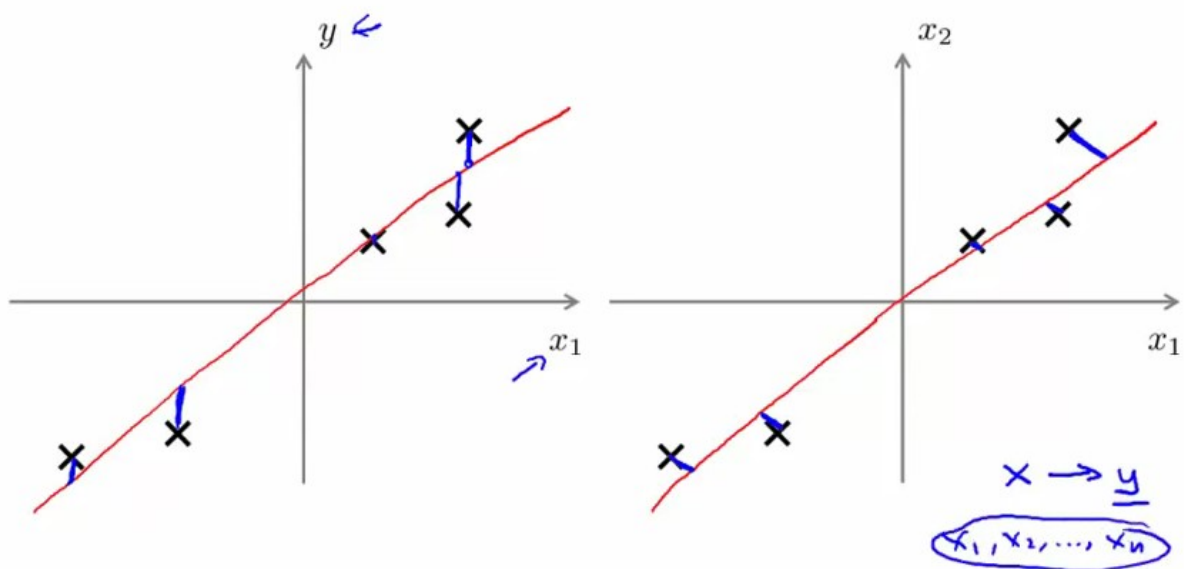
(<http://blog.csdn.net/linuxcunt>)

따라서 n 차원을 k 차원으로 축소할때는 각 데이터를 k 개의 벡터 u 에 대해 투영시켰을때의 *projection error* 를 최소로 하는 벡터 u 를 찾으면 된다.

Reduce from n -dimension to k -dimension, find k vectors u^1, \dots, u^k onto which to project the data, so as to minimize the projection error

쉽게 생각하면 k 개의 *direction* 을 찾는다고 생각하면 된다.

PCA is not linear regression



(<http://blog.csdn.net/linuxcunt>)

왼쪽 그림은 *linear regression* 에서 찾아내는 오차, 즉 y 값과 *prediction* 간의 거리이고

우측 그림은 *PCA* 로 찾아낸 선과 각 점사이의 *projection error* 를 파란색으로 나타냈다.

두 그림에서 볼 수 있듯이 *PCA* 는 *linear regression* 이 아니다. *PCA* 에서는 y 값이란 개념이 없다.

PCA Algorithm

Data preprocessing

Training set: $x^{(1)}, x^{(2)}, \dots, x^{(m)} \leftarrow$

Preprocessing (feature scaling/mean normalization):

$$\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)}$$

Replace each $x_j^{(i)}$ with $x_j - \mu_j$.

If different features on different scales (e.g., x_1 = size of house, x_2 = number of bedrooms), scale features to have comparable range of values.

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

(<http://blog.csdn.net/linuxcunt>)

PCA 전에는 *preprocessing step* 을 거친다. 이는 다양한 *feature* 간 스케일이 다르기 때문에 비교할만한 스케일을 얻기 위함이다.

Principal Component Analysis (PCA) algorithm

Reduce data from n -dimensions to k -dimensions

Compute "covariance matrix":

$$\Sigma = \frac{1}{m} \sum_{i=1}^n \underbrace{(x^{(i)})}_{n \times 1} \underbrace{(x^{(i)})^T}_{1 \times n}$$

Sigma

Compute "eigenvectors" of matrix Σ :

$$\rightarrow [U, S, V] = \text{svd}(\text{Sigma});$$

→ Singular value decomposition
Sig (Sigma)

$n \times n$ matrix.

$$U = \begin{bmatrix} | & | & | & \dots & | \\ u^{(1)} & u^{(2)} & u^{(3)} & \dots & u^{(m)} \\ | & | & | & & | \end{bmatrix}$$

$U \in \mathbb{R}^{n \times n}$

(<http://blog.csdn.net/linuxcumt>)

그 후에는 n 차원으로부터 *projection error* 가 최소인 k 개의 벡터를 얻는 계산을 수행한다.

- (1) 먼저 **Sigma** 라 부르는 *covariance matrix* 를 계산하고 (작은 시그마)
- (2) 그 후에 **Sigma** 의 *eigenvectors* 를 계산한다.

여기서 **svd** 는 *singular value decomposition* 을 의미한다.

Sigma 를 얻기 위한 *vectorization* 은 $(1/m) * X' * X$ 다.

Principal Component Analysis (PCA) algorithm

From $[U, S, V] = \text{svd}(\text{Sigma})$, we get:

$$\rightarrow U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}$$

$x \in \mathbb{R}^n \rightarrow z \in \mathbb{R}^k$

$$z = \begin{bmatrix} | & | & \dots & | \\ u^{(1)} & u^{(2)} & \dots & u^{(k)} \\ | & | & & | \end{bmatrix}^T \quad X^{(i)} = \begin{bmatrix} \text{---} (u^{(1)})^T \text{---} \\ \vdots \\ \text{---} (u^{(k)})^T \text{---} \end{bmatrix}$$

$z \in \mathbb{R}^k$ $U_{\text{reduce}} \quad n \times k$ $k \times n$ $k \times 1$

$X^{(i)} \quad n \times 1$

(<http://blog.csdn.net/linuxcumt>)

svd 함수의 리턴값으로 **U** 매트릭스가 나오는데, 이건 $n \times n$ 매트릭스다. 여기서 첫 k 개의 컬럼을 취한다. 이 매트릭스를 **Y** 라 부르면 새로운 *feature vector* **z** 는

$$z = Y^T * x$$

정리하면 아래 그림과 같다.

Principal Component Analysis (PCA) algorithm summary

→ After mean normalization (ensure every feature has zero mean) and optionally feature scaling:

$$\text{Sigma} = \frac{1}{m} \sum_{i=1}^m (x^{(i)})(x^{(i)})^T$$

→ $[U, S, V] = \text{svd}(\text{Sigma})$;

→ $\text{Ureduce} = U(:, 1:k)$;

→ $z = \text{Ureduce}' * x$;

↑

↑

$x \in \mathbb{R}^n$

~~$x_0 = 1$~~

$$X = \begin{bmatrix} - & x^{(1)T} & - \\ & \vdots & \\ - & x^{(m)T} & - \end{bmatrix}$$

$$\text{Sigma} = (1/m) * X' * X$$

(<http://blog.csdn.net/linuxcunt>)

PCA Details

넘어가기 전에 잠깐 PCA 에서 다룬 *covariance matrix* 나 *eigen vector* 를 좀 보고 넘어가자.

Covariance

두 확률 변수 X, Y 에 대해서 *covariance* 는

$$\begin{aligned} \sigma(X, Y) &= E[(X - E(X))(Y - E(Y))] \\ &= E(XY) - E(X)E(Y) \text{ (by linearity of expectation)} \end{aligned}$$

보면 알겠지만, 독립이면 *covariance* 값이 0 이다. 따라서 두 변수간 상관정도라 보면 된다. 공분산이 양수면 양의 상관관계를, 음수이면 음의 상관관계를 가진다.

이해를 위해서 [여기](#)로 부터 인용을 하자면

x의 분산은 x들이 평균을 중심으로 얼마나 흩어져 있는지를 나타내고, x와 y의 공분산은 x, y의 흩어진 정도가 얼마나 서로 상관관계를 가지고 흩어졌는지를 나타낸다. 예를 들어, x와 y 각각의 분산은 일정한데 x가 E(x)보다 클때

y도 $E(y)$ 보다 크면 공분산은 최대가 되고, x가 $E(x)$ 보다 커질때 y는 $E(y)$ 보다 작아지면 공분산은 최소(음수가 됨), 서로 상관관계가 없으면 공분산은 0이 된다.

공분산은 몇 가지 성질이 있는데

(1) X, Y 가 독립이면 $covariance = 0$ 이다. 그러나 역은 *gaussian random variable* 일때만 성립한다.

(2)

$$\text{Cov}(X, Y) = \text{Cov}(Y, X)$$

(3)

$$\text{Cov}(X, X) = \text{Var}(X)$$

(4)

$$\text{Cov}(aX, bY) = ab \text{ Cov}(X, Y)$$

공분산마다 값이 다르기 때문에 비교를 위해 X, Y 의 표준편차로 나눈 것을 *correlation*, *상관계수* 라 부른다.

$$\rho_{X,Y} = \text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y},$$

Covariance Matrix

이제 *covariance matrix* 를 알아보자. 공분산 행렬은 데이터 X (벡터) 에 대해 X 의 두 원소 $x^{(i)}, x^{(j)}$ 간 공분산을 구한 행렬이다. X 를 n 벡터라 하면, X 의 공분산 행렬은 $n \times n$ 행렬이다. 그리고 $\text{Cov}(X, Y) = \text{Cov}(Y, X)$ 이므로 *symmetric matrix*, *대칭행렬* 이기도 하다.

Principal Component Analysis (PCA) algorithm

Reduce data from n -dimensions to k -dimensions

Compute "covariance matrix":

$$\Sigma = \frac{1}{m} \sum_{i=1}^n (x^{(i)})(x^{(i)})^T$$

Sigma

Compute "eigenvectors" of matrix Σ :

$$[U, S, V] = \text{svd}(\text{Sigma});$$

→ Singular value decomposition
 $\text{eig}(\text{Sigma})$

$n \times n$ matrix.

$$U = \begin{bmatrix} | & | & | & \dots & | \\ u^{(1)} & u^{(2)} & u^{(3)} & \dots & u^{(m)} \\ | & | & | & & | \end{bmatrix}$$

$U \in \mathbb{R}^{n \times n}$

(<http://blog.csdn.net/linuxcumt>)

Eigen vector, Eigen value

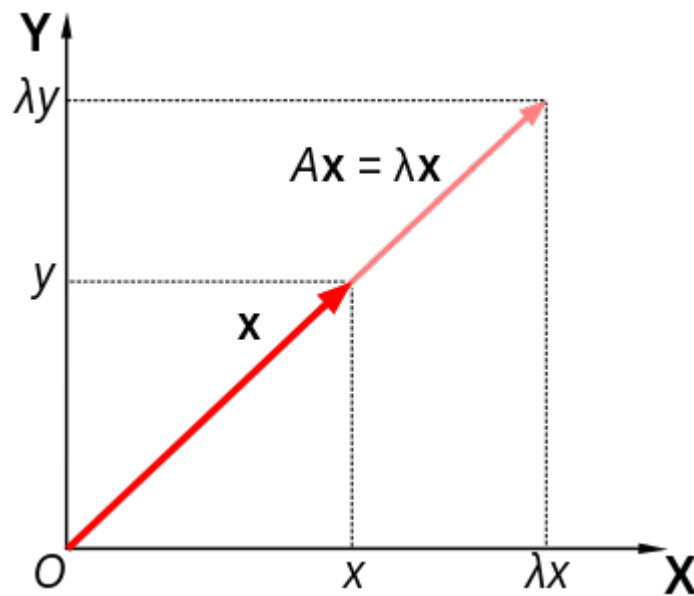
아까 슬라이드에서 잠깐 *eigen vector* 가 나왔는데, 우리말로 *고유벡터* 라 부른다. 고유 벡터 v 는 행렬 A 곱했을때 상수 λ 와 다음의 관계를 가진다. (A 는 $n \times n$ 매트릭스)

$$Av = \lambda v$$

이해를 위해 [여기](#)서 인용하면

square matrix A 를 선형변환으로 봤을 때, 선형 변환 A 에 의한 변환 결과가 자기 자신의 상수배 λ 가 되는 0 이 아닌 벡터 v 를 *eigen vector* 라 하고, 이 λ 를 *eigen value* 라 한다.

기하학적으로 보면



(<http://en.wikipedia.org>)

Diagonalization

SVD에 대해 이야기 하기 전에 *matrix diagonalization*, 행렬 대각화도 좀 보자.

대각행렬은 *principal diagonal* 원소를 제외한 모든 원소가 0인 *square matrix*인데, *square matrix*, 정방행렬 A 에 대해서

$$P^{-1}AP = D$$

인 P 와 D 가 존재하면 A 는 *diagonalizable matrix*, 대각화 가능 행렬 P 를 *diagonalizing matrix*, 대각화하는 행렬이라 부른다.

위 식에서 D 는 *diagonal matrix*인데,

$$D = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

저 식에서 A 위주로 정리하면

$$A = PDP^{-1}$$

이 때 P 가 고유벡터를 열벡터로 하는 행렬이고, D 가 고유값들을 대각 원소로 하는 대각행렬이면 *eigen decomposition* 이라 부른다.

위에서 *covariance matrix* 는 대칭행렬이라 말했는데, ($A = A^T$) 이 대칭행렬은

(1) 항상 *eigen decomposition* 이 가능하며

(2) *orthogonal matrix* 로 대각화가 가능하다. ($P^{-1} = P^T$)

Singular Value Decomposition

이제, *Singular Value Decomposition*, 특이값 분해에 대해 이야기 하자. *SVD* 도 행렬을 대각화 하는 한 방법인데, *eigen decomposition* 과 달리 $m \times n$ 행렬에 적용 가능하다.

$m \times n$ 행렬 A 에 대해 *SVD* 는

$$A = U\Sigma V^T$$

여기서 $m \times m$ 의 U 는 AA^T 를 *eigen decomposition* 해서 얻은 *orthogonal matrix* 고, U 의 열벡터를 A 의 *left singular vector* 라 부른다.

이 때 U *eigen decomposition* 해서 나온 *diagonalizing matrix* 이므로 U 의 열벡터는 AA^T 의 *eigen vectors* 다.

$$AA^T = U(\Sigma\Sigma^T)U^T$$

$n \times n$ 의 V 는 A^TA 를 *eigen decomposition* 해서 얻은 *orthogonal matrix* 고 V 의 열벡터를 A 의 *right singular vector* 라 부른다.

$$A^TA = V(\Sigma^T\Sigma)V^T$$

마찬가지로 V 도 *eigen decomposition* 의 결과로 얻은 *diagonalizing matrix* 이므로 V 의 열벡터는 $A^T A$ 의 *eigen vectors* 다

Σ 는 $AA^T, A^T A$ 를 *eigen decomposition* 해서 얻은 *eigen value* 의 *square root* 를 대각원소로 하는 $m \times n$ 의 직사각 대각행렬이다. 이 때 대각 원소들이 A 의 *singular value*, 특이값이다.

$$\Sigma = \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_s \\ & & & 0 \end{pmatrix} (m > n) \text{ or } \Sigma = \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_s & 0 \end{pmatrix} (m < n)$$

(관련 그림과 설명은 [여기](#)서 참조했습니다.)

이 때 AA^T 와 $A^T A$ 의 고유값 λ 는 동일하며 0 이상이다. 그렇기 때문에 *square root* 를 씌우고, 동일한 행렬 Σ 로 표현할 수 있다. (자세한 설명은 [여기](#) 참조)

AA^T 와 $A^T A$ 의 공통의 고유값에 대해

$$\sigma_1^2 \geq \sigma_2^2 \geq \dots \geq \sigma_s^2 \geq 0 \quad (s = \min(m, n))$$

이고 여기에 *square root* 를 취한 것이 A 의 *singular value*, 특이값이며, 이 특이값들을 대각원소로 하는 $m \times n$ 행렬이 Σ 다.

이 때 A 의 특이값과 *left singular value* u_i , *right singular value* v_i 에 대해

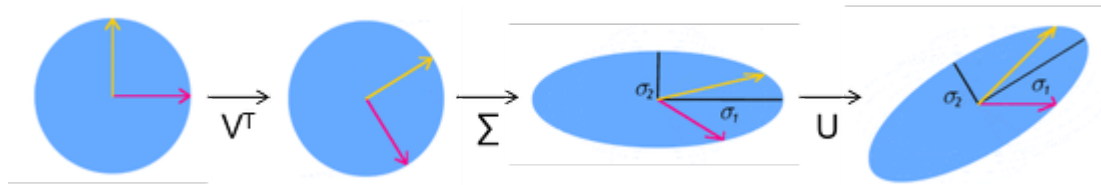
$$Av_i = \sigma_i u_i$$

SVD 의 기하학적 의미는 [여기](#)를 인용하면

행렬을 $x' = Ax$ 와 같이 좌표공간에서의 선형변환으로 봤을 때 직교행렬 (orthogonal matrix)의 기하학적 의미는 회전변환(rotation transformation) 또는 반전된(reflected) 회전변환, 대각행렬(diagonal matrix)의 기하학적 의미는 각 좌표성분으로의 스케일변환(scale transformation)이다.

행렬 R 이 직교행렬(orthogonal matrix)이라면 $RR^T = E$ 이다. 따라서 $\det(RR^T) = \det(R)\det(R^T) = \det(R)^2 = 1$ 이므로 $\det(R)$ 는 항상 $+1$, 또는 -1 이다. 만일 $\det(R)=1$ 라면 이 직교행렬은 회전변환을 나타내고 $\det(R)=-1$ 라면 뒤집혀진(reflected) 회전변환을 나타낸다.

따라서 식 (1), $A = U\Sigma V^T$ 에서 U, V 는 직교행렬, Σ 는 대각행렬이므로 Ax 는 x 를 먼저 V^T 에 의해 회전시킨 후 Σ 로 스케일을 변화시키고 다시 U 로 회전시키는 것임을 알 수 있다.



(<http://darkpgmr.tistory.com/106>)

다시 처음의 슬라이드로 돌아가면

Principal Component Analysis (PCA) algorithm

Reduce data from n -dimensions to k -dimensions

Compute "covariance matrix":

$$\Sigma = \frac{1}{m} \sum_{i=1}^n \underbrace{(x^{(i)})}_{n \times 1} \underbrace{(x^{(i)})^T}_{1 \times n} \quad \text{Sigma} \quad n \times n$$

Compute "eigenvectors" of matrix Σ :

$$\rightarrow [U, S, V] = \text{svd}(\text{Sigma}); \quad \text{Singular value decomposition} \quad \text{eig}(\text{Sigma}) \quad n \times n \text{ matrix.}$$

$$U = \begin{bmatrix} | & | & | & \dots & | \\ u^{(1)} & u^{(2)} & u^{(3)} & \dots & u^{(m)} \\ | & | & | & \dots & | \end{bmatrix} \quad U \in \mathbb{R}^{n \times n}$$

(<http://blog.csdn.net/linuxcunt>)

여기서 `svd` 함수의 인자 `Sigma` 가 *covariant matrix* 고 리턴값 `U, S, V` 가 각각 위에서 본 `U, \Sigma, V` 다. ~~변수 이름을 헛갈리게 지으심;~~

마지막 질문이다. *PCA* 에서 데이터 매트릭스 `X` 에 대해 *covariant matrix* `XXT` 로 구한 *eigen vector* 의 열벡터가, 왼쪽부터 순서대로 분산을 최대로 하는 벡터(방향)인데, ([여기](#)참조)

왜 우리는 *SVD* 의 `U` 를 택하는 것일까? 다시 말해 *PCA* 와 *SVD* 는 무슨 관계일까?

[What is the intuitive relationship between SVD and PCA](#) 를 참조하면,

데이터 매트릭스 `X` 에 대해서, 공분산 매트릭스 `XXT` 에 대해

$$XX^T = WDW^T$$

이 때 `X` 의 *SVD* 는

$$X = U\Sigma V^T$$

`U, V` 는 위에서 언급했듯이 `XXT, XTX` 의 고유값 분해로 얻은 대칭행렬 이므로 `VVT = I, UUT = I` 이다. 따라서

$$\begin{aligned} XX^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\ &= (U\Sigma V^T)(V\Sigma U^T) \\ &= U\Sigma^2 U^T \end{aligned}$$

이므로 분산이 큰 순서대로의 벡터를 열벡터로 담고 있는 `XXT = WDWT` 에서의 `W` 가 바로 *SVD* 의 결과인 `U` 다.

실제로 *PCA* 를 하기 위해 *SVD* 를 이용하는건 *numerically* 더 낫다고 한다.

In fact, using the SVD to perform PCA makes much better sense numerically than forming the covariance matrix to begin with, since the formation of XX^T can cause loss of precision. This is detailed in books on numerical linear algebra, but I'll leave you with an example of a matrix that can be stable SVD'd, but forming XX^T can be disastrous

다 정리하고 보니 드는 생각이,

“왜 공분산 행렬을 *eigen decomposition* 한 결과 $XX^T = WDW^T$ 에서 W 가 공분산 행렬, 즉 데이터간 상관관계, 즉 데이터 그 자체를 설명하는 걸까?”

여기 에서 얻은 답은,

공분산 매트릭스 자체는 *diagonal matrix* 가 아니다. 다시 말해 데이터 X 의 각 변수간 상관 관계를 담고 있다.

그런데, 공분산 매트릭스를 대각화 한다면 변수간 상관관계는 사라진다. 다시 말해

$XX^T = WDW^T$ 에서 D 자체에는 본래 데이터 X 의 변수간 상관 관계가 포함되어 있지 않다. 그러면, 그 데이터는 다 W 와 W^T 에 들어있다는 소리인데, W^T 는 W 로 표현 가능하므로 W 에 데이터간 상관 관계가 모두 담겨있다는 소리다.

Covariance matrix C_y (it is symmetric) encodes the correlations between variables of a vector. In general a covariance matrix is non-diagonal (i.e. have non zero correlations with respect to different variables).

But it's interesting to ask, is it possible to diagonalize the covariance matrix by changing basis of the vector?. In this case there will be no (i.e. zero) correlations between different variables of the vector.

Diagonalization of this symmetric matrix is possible with eigen value decomposition.

그러면 마지막 질문, W 의 좌측열부터가 왜 높은 분산을 가질까?

Choosing the number of PCA

적절한 k 의 수는 어떻게 구할까?

Choosing k (number of principal components)

Average squared projection error: $\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2$

Total variation in the data: $\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$

Typically, choose k to be smallest value so that

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq \frac{0.01}{0.10} \quad \frac{(1\%)}{(10\%)}$$

> “99% of variance is retained”
95% to 90%

(<http://blog.csdn.net/linuxcunt>)

위에서 언급했듯이 PCA 가 하는 일은 *projection error* 를 최소화 하는 것이다.

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01$$

인 k 를 구하면 99% 의 *variance* 가 유지된다. 적당한 *threshold* 값에 해당하는 k 값을 찾으면 된다.

알고리즘은 이런데 (좌측),

Choosing k (number of principal components)

Algorithm:

Try PCA with $k=1$ ~~$k=2$~~ ~~$k=3$~~ $k=4$

Compute $\sigma_1, \dots, \sigma_k$

$\rightarrow [U, S, V] = \text{svd}(\text{Sigma})$

$S = \begin{bmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \ddots \end{bmatrix}$

compute $\mu_{reduce}, \dots, \mu_{approx}, \dots, x_{approx}^{(1)}, \dots, x_{approx}^{(m)}$

Check if

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01?$$

$k=17$

For given k

$$1 - \frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \leq 0.01$$

$$\frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \geq 0.99$$

Andreas B

(<http://blog.csdn.net/linuxcunt>)

매번 계산하는건 굉장히 비 효율적이다. 따라서 우측처럼 데이터에 대한 *singular value* 를 이용하면 더 계산이 쉬워진다.

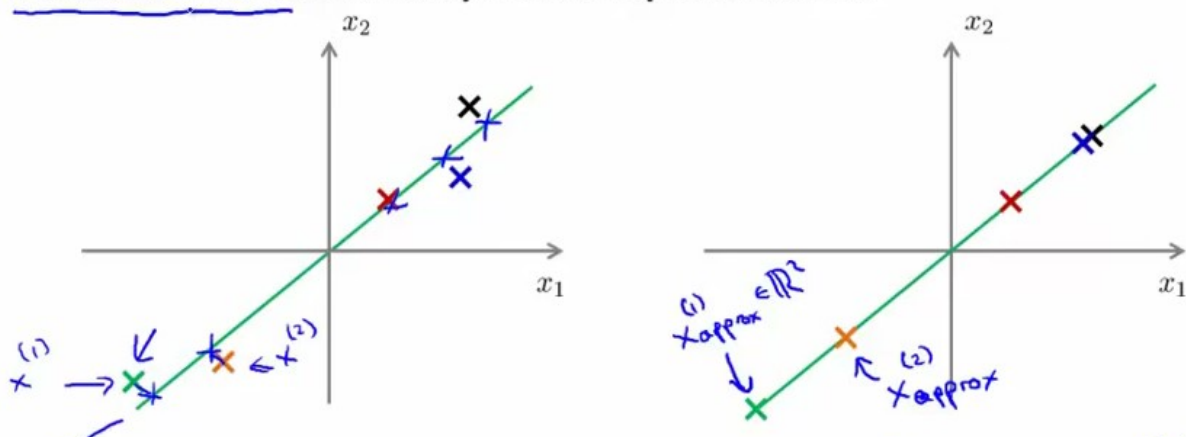
$$\frac{\sum_{i=1}^k}{\sum_{i=1}^n} \geq 0.99$$

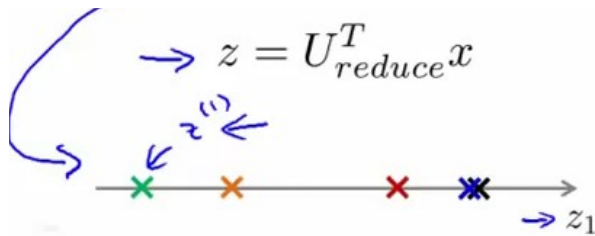
Reconstruction from Compressed Representation

잘 보면 *PCA* 가 하는 일은 높은 차원의 데이터를 최대한 보존하면서 차수를 줄이는 일이다. 압축 알고리즘과 비슷하다. (실제로 이미지 압축에 쓴다고 한다.)

그럼 압축된 차원 z 에서 다시 본래의 데이터 차원 x 를 복구하려면 어떻게 해야할까?

Reconstruction from compressed representation





$$z \in \mathbb{R} \rightarrow x \in \mathbb{R}^c$$

$$\begin{bmatrix} \tilde{x}^{(1)} \\ \vdots \\ \tilde{x}^{(m)} \end{bmatrix}_{\mathbb{R}^n} = \underbrace{U_{reduce}}_{n \times k} \cdot \underbrace{z^{(1)}}_{k \times 1}$$

Diagram illustrating the reconstruction of x from z using the matrix U_{reduce} . The dimensions are $n \times k$ for U_{reduce} and $k \times 1$ for $z^{(1)}$, resulting in $n \times 1$ for $\tilde{x}^{(1)}$.

(<http://blog.csdn.net/linuxcunt>)

위 그림처럼 $z = U^T * x$ 라 할때 좌변에 U 를 곱하면 $x_{app} = U * z$ 에서 x_{app} 는 거의 x 에 가까워진다.

Advice for Apply PCA

Supervised learning speedup

→ $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

Extract inputs:

Unlabeled dataset: $x^{(1)}, x^{(2)}, \dots, x^{(m)} \in \mathbb{R}^{10000}$

↓ PCA

$z^{(1)}, z^{(2)}, \dots, z^{(m)} \in \mathbb{R}^{1000}$

New training set: $(z^{(1)}, y^{(1)}), (z^{(2)}, y^{(2)}), \dots, (z^{(m)}, y^{(m)})$

Note: Mapping $x^{(i)} \rightarrow z^{(i)}$ should be defined by running PCA only on the training set. This mapping can be applied as well to the examples $x_{cv}^{(i)}$ and $x_{test}^{(i)}$ in the cross validation and test sets.

Handwritten notes and diagrams:

- $x^{(i)} \in \mathbb{R}^{10,000}$ (with a 100x100 box representing the input space)
- U_{reduce} (handwritten label for the PCA transformation)
- $h_\theta(z) = \frac{1}{1 + e^{-\theta^T z}}$ (handwritten sigmoid function)
- $x \rightarrow z$ (handwritten mapping)

(<http://blog.csdn.net/linuxcunt>)

supervised learning의 속도를 올리는데도 쓸 수 있다. 이미지가 100×100 이면 10000 개의 feature 인데, 이건 어마어마하다.

먼저 input x 를 뽑아내 여기에 대해 PCA 를 실행하면 차원을 줄인 training set 을 얻을 수 있다.

주의할점은 U 를 찾을때 *training set*에만 하고 *cross validation* 이나 *test* 까지 포함해서 U 를 찾으면 안된다. 나중에 *training set* 으로만 찾아낸 U 를 이용해서 *CV*, *test* 에 대해 다시 *PCA* 하자.

다른 *PCA* 응용으로는

- Reduce memory, disk needed to store data
- Speed up learning algorithm
- Visualization ($k = 2$ or 3)

Bad use of PCA: To prevent overfitting

→ Use $z^{(i)}$ instead of $x^{(i)}$ to reduce the number of features to $k < n$. — 1000

Thus, fewer features, less likely to overfit.

Bad!

This might work OK, but isn't a good way to address overfitting. Use regularization instead.

$$\rightarrow \min_{\theta} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2 \quad \leftarrow$$

(<http://blog.csdn.net/linuxcunt>)

PCA 를 이용하면 *feature* 의 수가 줄기 때문에 *overfitting* 을 방지하기 위해 사용할 수 있다고 생각하겠지만, 별로 좋은 생각은 아니다.

작동은 할지 모르겠지만 *regularization* 을 이용하는 편이 낫다.

왜냐하면 *PCA* 는 y 값이 없는 상태에서 작동하기 때문에 y 를 고려하지 않은 데이터가 손실이 발생할 수 있다. 1% 만 손실된다 하더라도, 그 1% 가 y 와 관련해 굉장히 중요한 정보일 수 있다.

PCA is sometimes used where it shouldn't be

PCA IS SOMETIMES USED WHERE IT SHOULDN'T BE

Design of ML system:

- - Get training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
- - ~~Run PCA to reduce $x^{(i)}$ in dimension to get $z^{(i)}$~~
- - Train logistic regression on $\{(\cancel{z^{(1)}}), y^{(1)}), \dots, (\cancel{z^{(m)}}), y^{(m)})\}$
- - Test on test set: Map $x_{test}^{(i)}$ to $z_{test}^{(i)}$. Run $h_{\theta}(z)$ on $\{(z_{test}^{(1)}, y_{test}^{(1)}), \dots, (z_{test}^{(m)}, y_{test}^{(m)})\}$

→ How about doing the whole thing without using PCA?

→ Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want, then implement PCA and consider using $z^{(i)}$.

(<http://blog.csdn.net/linuxcumt>)

또 다른 잘못된 PCA의 사용으로는, 그냥 무작정 PCA를 사용하는 것이다.

original data x 에 대해 알고리즘을 구현도 안해보고, 바로 PCA의 결과인 z 를 이용하려는건 좋은 생각이 아니다.

x 에 대해 작업 해보고 결과가 별로일때 PCA를 고려하자.

References

- (1) *Machine Learning* by Andrew NG
- (2) <http://blog.csdn.net/linuxcumt>
- (3) <http://blog.csdn.net/abcjennifer>
- (4) <http://scipy-lectures.github.io>
- (5) Wiki: Correlation and dependence
- (6) <http://en.wikipedia.org/wiki/Covariance>
- (7) <http://darkpgmr.tistory.com/110>
- (8) <http://darkpgmr.tistory.com/105>
- (9) Wiki: Eigenvalues and Eigenvectors
- (10) <http://www.ktword.co.kr>
- (11) <http://darkpgmr.tistory.com/106>
- (12) What is the intuitive relationship between SVD and PCA

0 Comments lambda

 Login ▾

 Recommend  Share

Sort by Best ▾






Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Be the first to comment.

 Subscribe  Add Disqus to your siteAdd DisqusAdd  Privacy

comments powered by Disqus

